TEMPLATE NUMERICAL TOOLKIT FOR LINEAR ALGEBRA: HIGH PERFORMANCE PROGRAMMING WITH C++ AND THE STANDARD TEMPLATE LIBRARY

ROLDAN POZO *

Abstract. We present a new C++ library design for linear algebra computations on high performance architectures. The Template Numerical Toolkit (TNT) for Linear Algebra is a successor to the Lapack++, Sparselib++, and IML++ packages, providing support for direct and iterative solvers. Its goal is to formally integrate these ideas into a generic algorithm library supporting user-defined data types and data neutrality. The design of the core library utilizes components from the C++ Standard Template Library (STL) and the basic parallel extensions defined in HPC++.

1. Introduction. The *Template Numerical Toolkit* (TNT) for linear algebra is a successor to the Lapack++ [6], Sparselib++ [13], IML++ [5], and MV++ [12] packages. Its goal is to formally integrate these ideas into a generic algorithmic library, supporting user-defined data types, and increasing its functionality. The toolkit provides an integrated collection of generic matrix/vector classes based on components of the Standard Template Library (STL), together with specialization of generic algorithms for maximal efficiency. The TNT project is under development; this paper provides a look at the work in progress. Here we focus on issues of sparse matrices and iterative methods for the solution of linear systems.

The fundamental goal in TNT is to be able to express numerical algorithms *independent* of the specific matrix or vector implementation. For example, the *same* generic algorithm for a conjugate gradient algorithm can work for single-precision dense matrix, or various double-precision sparse matrix schemes. This is accomplished using consistent interfaces and template functions in C++.

Parallelism in TNT can be expressed in three ways: (1) utilizing threads for the Basic Linear Algebra Subprogram (BLAS) calls, (2) utilizing a shared-memory version of STL, as prescribed in HPC++[9], and (3) utilizing distributed vectors in a SPMD setting. This paper focuses on the first two methods. This type of SMP parallelism is not aimed at massively parallel architectures with thousands of processors, but is much more practical to integrate with existing application codes.

Another important aspect of the toolkit is that it provides different implementation choices for the basic linear algebra structures. For example, it supports four different variations of sparse vectors: a linked-list implementation (providing O(1) insertion, O(N) random-access cost), a vector-based implementation (providing fast random-access, but O(N) insertion cost), an associative-table implementation (providing efficient iteration mechanism, O(1) insertion cost, and O(log N) random-access cost), and a Fortran-compatible implementation (providing compatible data structures for external libraries and application codes). Furthermore, several standard sparse matrix storage schemes are supported, providing the application programmer with the greatest design flexibility in choosing the most appropriate storage-scheme for their application or architecture.

Nevertheless, many of the algorithms in TNT can be used on multiple linear algebra objects. This works not only for double-precision, complex, and user-defined types, but also for linked-list, vector, and map-based versions of sparse storage schemes. (Section 2 illustrates how this is done.) The resulting code reduction is a tremendous simplification over conventional schemes in which a separate function or program must be written for each storage-structure/data-element combination.

Adding two sparse matrices is as straightforward as "`A+B`", even if their structures are not identical. This is somewhat similar to the style used in Matlab, Mathematica, and other high-level scientific environments. One can also define new user-derived matrix types, such as a compressed-column matrix of 4x4 blocks, or a banded matrix of multi-precision numbers. Elements of a sparse matrix can be accessed as `A(i,j)`, regardless of its underlying representation or matrix storage format. Converting from one format to another can be as simple as "`A=B`".

* Applied and Computational Mathematics Division, National Institute of Standards and Techology,

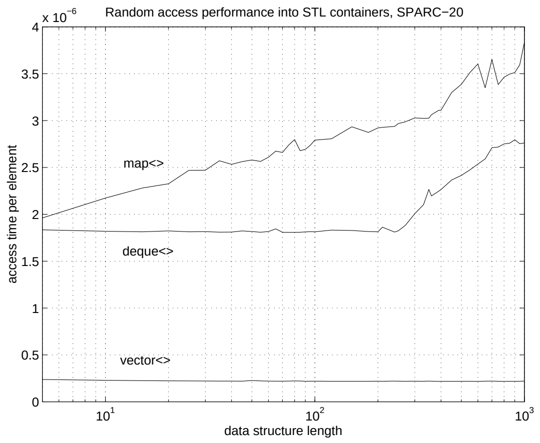

FIG. 1. *Random access performance of STL container classes on Sun SPARC 20, using g++ 2.7.2 with -O optimization.*

While elegant, such an approach could lead to inefficient C++ implementations if it was the *only* mechanism available to describe such algorithms. TNT provides various levels of the same algorithms, from the elegant data-independent descriptions to the low-level computational kernels. For example, optimized routines for the matrices of common data types (single-precision, double-precision, complex) will link with the Level 3 Sparse Blas[8],[3], [14] effort. Such an offering of algorithms at various levels of abstraction, provides a realistic tradeoff between program simplicity and performance, and is one of the new features in this library.

2. Data structures in TNT.

The vectors and matrices in TNT are built out of standard components. ANSI C++ defines a basic toolkit of reusable components useful for general programming. The *Standard Template Library* (STL) defines basic data structures such as queues, lists, sets, and algorithms for using these in higher-level applications.

STL provides reusable container, iterators, algorithms, and function objects. For designing vector and matrices for numeric programming, some of the most useful components are the `vector<>`, `list<>`, and `map<>` containers (shown in Fig. 3), together with their respective algorithms. The `vector<>` is similar to a C array (elements are stored contiguously in memory) except that one can "insert" an element into the middle of it. The `vector<>` structure will grow if there is enough room, or reallocate a larger memory segment and recopy itself into the new location. Element access time is constant, and typically as efficient as native C array indexing. Insertions and deletions are $O(N)$ if in the middle, or $O(1)$ if at the end of the vector. There is no support for index bounds checking. The `list<>` container is a doubly-linked list structure. Elements can be inserted or removed in constant time. Element lookup is $O(n)$. The `map<>` container is an associative-array structure with $O(\log n)$ element lookup cost, $O(1)$ and insertion/deletion cost.

Each container provides performance and efficiency tradeoffs in indexing and inserting. These are summarized by the following table:

	iteration cost	random cost	insert cost
map	8x	O(log N)	O(1)
vector	1x	O(1)	O(N)
list	5x	O(N)	O(1)

For these measurements we constructed containers of varying lengths and measured the time required to perform random-accesses into these structures.

2.1. Dense vectors and matrices.

TNT supports the vectors and matrices shown in Fig. 2. The basic numerical `Vector<>` class is closely based on the STL generic `vector<>` class, but adds

FIG. 2. *Linear algebra containers in TNT*

- dense vectors
  - `Vector<>`
- dense matrices
  - `C_Matrix<>`
  - `Fortran_Matrix<>`
- native sparse vectors
  - `sparse_vector<>`
  - `STLmap_sparse_vector<>`
  - `STLvec_sparse_vector<>`
  - `STLlist_sparse_vector<>`
- Fortran-compatible sparse vector
  - `Fortran_sparse_vector<>`
- native compressed sparse matrices
  - `Compressed_Sparse_Row_matrix<>`
  - `Compressed_Sparse_Col_matrix<>`
  - `Coordinate_Sparse_matrix<>`
  - `Compressed_diagonal_sparse_matrix<>`
- Fortran-compatible sparse matrices
  - `Fortran_Compressed_Sparse_Row_matrix<>`
  - `Fortran_Compressed_Sparse_Col_matrix<>`
  - `Fortran_Coordinate_Sparse_matrix<>`

several features for numerical computing, such as optional index bound checking (turned on or off at compile time via `TNT_BOUNDS_CHECK` macro), and indexing via parentheses: `x(i)`, as well as square brackets: `x[i]`.

The `C_Matrix<>` class mimics the behavior of C arrays, except that their sizes need not be specified at compile time. A `C_Matrix<>` object has for each row an independent vector. Thus, elements in each row can be treated as being contiguous in memory, but adjacent rows need not be. Indexing into a `C_Matrix<>` can be written as `A[i][j]` or `A(i,j)`. The expression `A[i]` denotes the ith row and can be used anywhere a vector can. The overhead in indexing a `C_Matrix<>` is no more than indexing a native C array.

The `Fortran_Matrix<>` class mimics the behavior of a two-dimensional Fortran array. Elements are column-oriented and contiguous in memory. For indexing efficiency, an MxN `Fortran_Matrix<>` has an extra pointer array of size N declared. Because all of the elements are contiguous, a `Fortran_Matrix<>` layout may be seen as a one-dimensional container and hence an STL container. Thus, any STL algorithm, such as `for_each()`, or `max()` may be used on objects of this class. The overhead in indexing a `Fortran_Matrix<>` is typically the same as a native C array.

Although each STL container defines a `::size_type()` indexing type, TNT vectors and matrices utilize the same index type: `Subscript`. This makes it easier to write expressions involving matrices and vectors.

2.2. Sparse Vectors. Sparse vectors are represented in TNT as generic containers of [value,index] pairs. If `ptr` is a pointer to a sparse vector entry, then `sp_value(*ptr)` returns that element's index, and `sp_value(*ptr)` returns that element's value. By keeping these two attributes together in a single structure, we can treat these sparse vector container as generic STL containers which can utilize the basic STL algorithms.

Figure 3 illustrates the TNT sparse vector data structures. The `STLlist_sparse_vector<>` class uses a doubly-linked list of [value, index] pairs. This provides an $O(1)$ insertion capability, but lacks random-access. The `STLvec_sparse_vector<>` class is much more efficient for traversing its elements, but exhibits linear $O(nz)$ complexity to find a specific element. The `STLmap_sparse_vector<>` class uses an associate table (STL map) container to hold the elements sorted by index value. This reduces the lookup complexity from $O(nz)$ to $O(\log nz)$; however, the cost of iterating through its elements

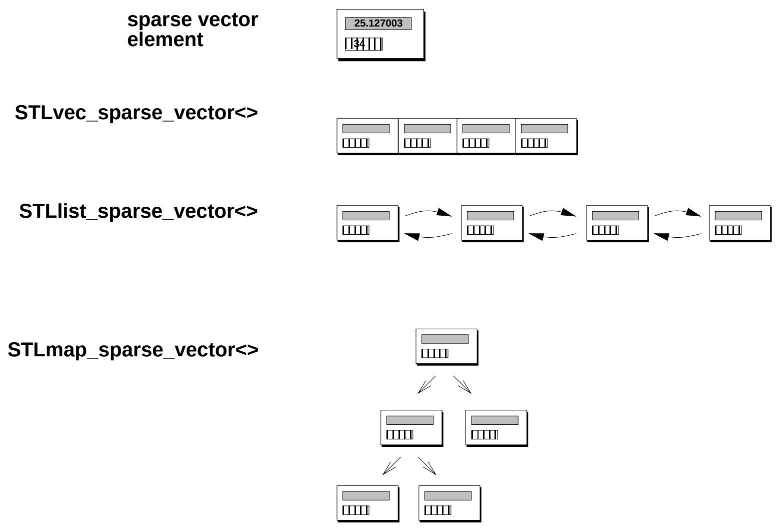

FIG. 3. *TNT data structures for sparse vectors.*

can be up to eight times slower than an STL `vector`. Finally, a Fortran-compatible storage scheme, `Fortran_sparse_vector<>`, utilizes two separate vectors: one for values, one for indices.

2.3. Sparse Matrices. TNT supports several sparse matrix formats, mainly compressed schemes built from sparse vector data objects described in the previous section. The `Compressed_Sparse_Row_matrix<>` and `Compressed_Sparse_Column_matrix<>` are one-dimensional collections of the native TNT sparse vectors.

To provide a simple illustration of how these matrices are used, we present an example which initializes a matrix from an input file, performs a matrix/vector multiply with it, and displays the result. First, let us assume the input matrix is stored as a coordinate text file format in which each nonzero `a(i,j)` is sorted on a separate text line as "i j a" . The first line shows the size of the matrix and the total number of nonzeros, e.g. a 5x5 sparse matrix with 4 nonzeros would look like

```
5 5 4
1   1   3.2
2   2   1.7
3   3   2.1
2   1   1.8
```

(Such a format loosely corresponds to the Matrix Market[2] exchange format, so it is not completely artificial.) Given such an input file, the code fragment to perform the initialization and computation looks like:

```
cin >> M >> N >> nz;            // read matrix size and nonzeros

Sparse_matrix A(M,N);           // declare sparse matrix

for (Subscript k=0; k<nz; k++)
{
    cin >> i >> j >> val;       // read in values
    A(i,j) = val;               // fill sparse matrix
}
```

- Kernel algorithms
  - algebraic operators (A+B, A+=B, A*B, etc.)
  - $y \rightarrow Ax$ matrix/matrix, matrix/vector products
  - $Lx = y$ triangular solves
  - $||A||$ various vector/matrix norms
- Direct methods
  - LU
  - QR
  - LL^T
- Eigenvalues
- Iterative methods
  - Richardson iteration
  - Chebyshev Iteration
  - Conjugate Gradient (CG)
  - Conjugate Gradient Squared (CGS)
  - BiConjugate Gradient Stabilized (BiCGSTAB)
  - Generalized Minimum Residual (GMRES)
  - Quasi-minimal Residual (QMR) without lookahead

FIG. 4. *Linear Algebra algorithms in TNT*

```
Vector x(M, 1);                 // x = [1, 1, ... ];

cout << A*x ;                   // perform mat/vec multiply
                                // and display results
```

The `Sparse_matrix` and `Vector` typenames can refer to any of the native TNT objects described in Fig. 2, by declaring typename aliases such as

```
typedef STLvec_sparse_vector<double>  Sparse_vector;
typedef Compressed_Sparse_Row_matrix<Sparse_vector> Sparse_matrix;
```

One could implement such as code fragment as a templated function in which the arguments are returned, rather than printed out.

3. TNT algorithms.

3.1. Iterative Methods.

TNT supports various direct and iterative methods, show in Fig. 3. These range from low level BLAS-like operations, such as matrix/vector multiplication, to high-level iterative methods, such preconditioned conjugate gradient methods. In this paper, we will focus on a few sample codes to illustrate some of the basic principles of TNT algorithms.

The advanced interfaces in TNT employ split-phase computation (i.e. a separate initialization and iteration section) for greater efficiency and flexibility. The efficiency comes from the fact that often one need not perform a termination test (usually some residual norm computation) at every iteration – particularly in the early stages of the iterations. By separating these two phases, one can forward the iteration several steps at a time, without calling the termination section. This technique moves the decision of when or how often to apply termination test out of the library and into the calling application, where it belongs.

Integrating the user-defined preconditioner and matrix/vector multiply routine with the internal iteration algorithms also poses challenges, particularly in strongly-typed languages like ANSI C and C++. Often these functions are provided as C or Fortran routines: `f(x, t1, t2, ... )` and specifying such an interface in a general library routine of an iterative methods package is impractical. Nevertheless, C++ *function objects* can provide a type-safe solution, without unnecessary copying or global variables.

The split-phase approach treats the iterative method as a conventional C++ class. The constructor is used to set the internal variables in the iterative algorithm, while the `::iterate()` method is used to advance the algorithm.

Consider a preconditioned conjugate-gradient example. The basic iteration code looks like:

```
template <class Matrix, class Vector, class Precond>
void CG_method<Matrix, Vector, Precond>::iterate()
{
    z = M(r);

    if (num_iters == 0) p=z;
    else
    {   beta = rho / dot_product(p,q);
        p = beta * p + z;
    }

    q = A*p;
    alpha = rho / dot_product(p,q);

    x += alpha * p;
    r -= alpha * q;

    rho1 = rho;
}
```

Notice that no mention is made of the preconditioner, nor of the specific matrix type (it could be sparse, dense, or distributed), nor the element types (single or double precision). Furthermore, in the case where the matrix is sparse, no mention is made of the particular storage format.

The constructor for the method requires the initial vector, the resulting solution, and the matrix and preconditioners as function objects. (Neither the matrix nor the preconditioner need be explicitly formed.) The constructor also initializes the internal variables:

```
template <class Matrix, class Vector, class Precond>
CG_method<Matrix, Vector, Precond>::CG_method(const Matrix &A,
    const Vector x0, const Vector &b, const Precond &M)
{
    x = x0;
    r = A*x - b;
    alpha = beta = rho = rho_1 = 0;
    num_iters = 0;
}
```

In this case, we have described a completely generic algorithmic description of the conjugate gradient algorithm which closely matches its mathematical description. The kernel operations, such as the `A*x` and `dot_product()` are then instantiated with specific data objects to determine which version they will link with this. This mechanism is described in the next section.

3.2. Computational kernels. At some point in the description of high-level algorithms, such as the iterative methods described above, data-specific versions of computational kernels must be instantiated to produce executable code. For example, a conjugate gradient algorithm instantiated with a sparse matrix needs to find the appropriate matrix-vector multiply routine to compute `A*x` in its inner loops. This is where the real work gets done.

Each subclass of matrices (e.g. sparse, dense, distributed dense, etc.) will have its own particular implementation of `A*x`. Although it is impossible to have one generic algorithm for each case, one

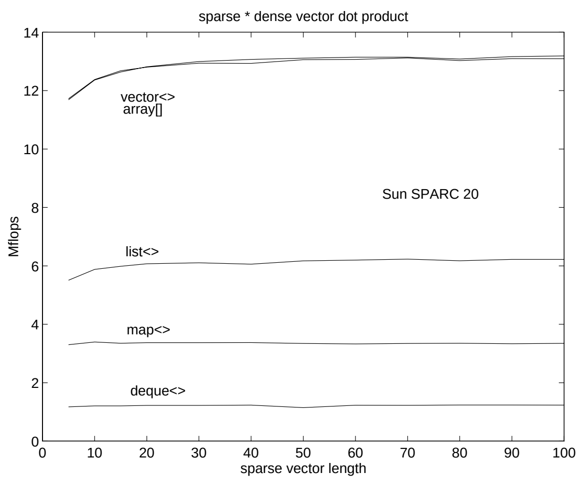

FIG. 5. *Dot product performance of various TNT sparse vector representations.*

can group similar TNT data structures together and exploit commonality to reduce the number of functions actually defined.

For example, the algorithm to perform a dot product of a a sparse vector with a dense vector:

```
template <class SparseInputIterator, class RandomAccessIterator, class T>
inline T  spvec_dot_product(SparseInputIterator p, SparseInputIterator last,
                    RandomAccessIterator x, T init)
{
    for (; p!=last; p++)
        init = init + sp_value(*p) * x[sp_index(*p)];
    return init;
}
```

will work with all three types of native TNT sparse vectors (Fig. 2. In each case, the algorithm walks through the nonzero elements in the sparse vector and uses each element's index (`sp_index()`) to extract the corresponding value from the dense vector `x`. Actually, `x` need not be a vector, but any container that support random access iterators. Figure 5 illustrates how the various performance rates for dot products between sparse and dense vectors compare.

The following code illustrates how a matrix/vector multiply algorithm ($y \leftarrow A * x + y$) can be written for generic compressed-row sparse matrices:

```
template <class CompressedSparseRowIterator, class RandomAccessIterator>
void inline csr_mat_vec_mult(CompressedSparseRowIterator p,
      CompressedSparseRowIterator last, RandomAccessIterator x,
        RandomAccessIterator y)
{
        while (p!=last)
        {
            *y = spvec_dot_product((*p).begin(), (*p).end(), x, *y);
            p++;
            y++;
        }
}
```

This algorithm can be used by any sparse matrix storage scheme which maintains its elements as a series of sparse vectors, one for each row. This includes not only the TNT `Compressed_Sparse_Row` container adaptor, but user-defined implementations as well. The only requirement is that the sparse vector container used must support iterators to move through its elements, as well the `sp_index()` and `sp_value()` access functions.

3.3. Conventional Sparse BLAS. For the common cases where the elements of matrix and vector elements are single or double precision numbers, TNT can link with optimized routines for the low-level numerical operations. For dense matrices, one widely used library is the Basic Linear Algebra Suprograms (BLAS), which has been implemented on a wide array of computer architectures. The BLAS library supports matrix/matrix multiplication routines, together with triangular solves. Similar kernels for sparse matrices are currently being developed. One current implementation, the NIST Sparse BLAS [14] library provides the following operations:

- sparse matrix products,

$$C \leftarrow \alpha\ A\ B + \beta C$$

$$C \leftarrow \alpha\ A^T\ B + \beta C$$

- solution of triangular systems,

$$C \leftarrow \alpha D_L\ A^{-1}\ D_R B + \beta C$$

$$C \leftarrow \alpha D_L\ A^{-T}\ D_R B + \beta C$$

where A is sparse matrix, B and C are dense matrices/vectors, and D_L and D_R are diagonal matrices. This version of the NIST Sparse BLAS supports the following sparse formats: compressed sparse row (CSR), compressed sparse column (CSC), coordinate (COO), block sparse row (BSR), block sparse column (BSC), block coordinate (BCO) and variable block row (VBR). Symmetric and skew-symmetric versions are also supported.

The routines are written in ANSI C and are callable from Fortran and C through the interface proposed in the **Sparse BLAS Toolkit**[3] Also see the companion paper [8]. Performance results for various computer architectures are shown in Fig. 6.

4. Parallelism. Shared-memory parallelism in TNT can be expressed in three ways: (1) using the shared-memory pragmas described in High Performance C++ (HPC++) [9], (2) replacing the conventional STL header files with parallel versions, and (3) using conventional thread-based parallelism in supporting libraries, such as the sparse and dense BLAS.

Each of these approaches has its strengths and weaknesses. For example, the HPC++ pragmas are some of the most powerful techniques, but the HPC++ design specification is still evolving and there is no compiler support as of this writing. A parallel implementation of the STL, on the other hand, can be quite challenging, particularly for containers that do not support random-access iterators. Finally, a thread-based solution for non-templated functions is possible within ANSI C++, using existing operating system interfaces, such as POSIX threads[16], or Windows NT threads[11].

4.1. HPC++ pragmas. The easiest and most powerful approach to express parallelism in TNT (and in fact most C++ programs) is to utilize parallel extensions specified by the HPC++ working group. Although this is an ongoing effort, we expect that compilers may support such constructs in the future, and include them here to illustrate how such an approach would work with TNT.

The use of pragmas in HPC++ is similar to the directives in High Performance Fortran (HPF). The basic idea is to annotate a conventional sequential program, identifying possible parallel loops. This annotation is in the form of comments, or pragmas in C++, which can be safely ignored by a conventional ANSI C++ compiler. Thus, the resulting code is still portable and can be run on sequential platforms without modification.

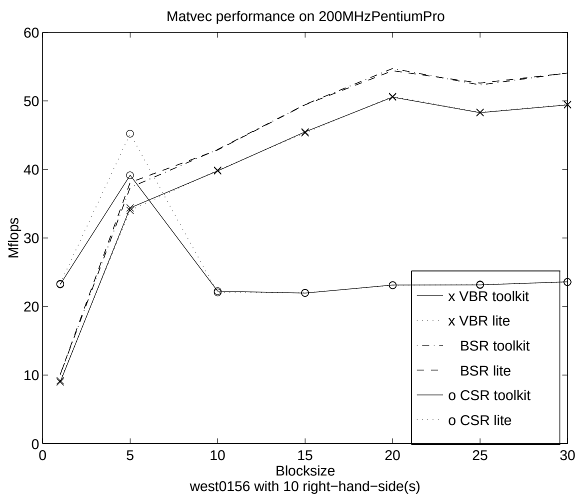

Fig. 6. *Performance of NIST Sparse BLAS matrix/multiply operation based on block-structure matrix with various storage schemes, on a 200 MHz Intel Pentium Pro (P6) workstation.*

HPC++ identifies well-behaved parallel loops via the HPC_INDEPENDENT pragma. For example, a code segment to multiply a sparse vector by a scalar in parallel, can be written as

```
template <class SparseVec, class Scalar>
SparseVec& spvec_mult_scalar_update(const SparseVec &A, const Scalar& a)
{
    #pragma HPC_INDEPENDENT
    for (SparseVec::iterator p=A.begin(); p != A.end() ; p++)
        sp_val(*p) *= a;

    return A;
}
```

If the iterator is a random-access type, an HPC++ compiler can transform this template declaration from iterator-based to index based. In this case, the values of the induction variable can be broken into separate threads. For containers without random-access, the situation becomes more difficult. (See section 4.2.)

For loops involving reduction variables, a similar HPC_REDUCE pragma may be used, as in

```
template <class SparseVec, class Vector, class scalar>
inline scalar spvec_dot_product( const SparseVec &A, const Vector &B,
            const scalar &prev_sum)
{   scalar sum = prev_sum;

    for (SparseVec::const_iterator p=A.begin(); p!=A.end(); p++)
        #pragma HPC_REDUCE
        sum += sp_value(*p)  *   B[sp_index(*p)];

    return sum;
}
```

to compute the dot product of sparse vector and a dense vector. The syntax of this pragma can also specify PRIVATE variables, as well as different reduction operations, such as min, max, "+", "-", "*", and so on.

The complete specification for both of these pragmas is much more complex. The HPC_INDEPENDENT directive is designed to support private variable lists, and ON_HOME(mapped_variable) extensions for distributed-memory programming. See the HPC++ Working Group White Paper[9] for further details.

4.2. Parallel versions of STL. The simplest approach is to apply HPC++ loop pragmas to the STL algorithms, creating parallel variations of for_each(), count(), and so on, as described in [10].

Aside from this automatic compiler parallelization, another solution is to implement a modified version of STL which has been parallelized with an existing thread interface, such as Windows NT or POSIX threads. This is very attractive to the application programmer, since "parallelizing" a code would involve only recompiling (with a different set of STL header files) and re-linking. Furthermore, it could be accomplished with unmodified ANSI C++ compilers and does not rely on experimental languages. (Operating systems that do not support threads would just see the conventional STL.) The assumption here, of course, is that the parallelism would be limited to sections of code using STL.

However, there is still a basic challenge with this approach: the algorithms in STL are *iterator*-based. That is, the basic scheme when traversing containers is to move from one element to the next. Most of the STL containers, such as list<> and set<>, do not support random-access — the only way to get to the i-th element is to have visited the (i-1)th element. This makes it very difficult, without detailed compiler analysis, to break up the iteration space into separate threads. Clearly, if one must first traverse the complete list sequentially to partition the list, the parallel benefit is seriously degraded.

One solution is to construct a modified STL container that maintains a list of markers that partitions the list into equal-sized chunks. However, this has its own problems.

First, since the STL container needs a variant implementation, one cannot utilize shared-memory parallelization "on the fly". That is, one cannot decide at a given point in the computation that a given container will be processed in parallel. Instead one has to decide apriori which objects will be parallel — often difficult to do when developing library code.

Second, it requires maintenance for the markers — every time an element is inserted or deleted one needs to re-adjust these markers so that the resulting partitions are "load-balanced". This can greatly increase the cost of insert/delete operations to the point where the increased overhead is greater than the benefit from processing the list in parallel. (This is not the case if the amount of work being done relative to the indexing cost is large.)

Third, the number of markers need to be determined at time of object construction, thus fixing the number of threads during the lifetime of the object.

4.3. OS threads and function pointers in ANSI C++. One practical solution for SMP parallelization with ANSI C++ is the use of Windows NT or POSIX threads. The former has enjoyed widespread interest, since multiprocessor workstations and servers based on Pentium Pro and Alpha processors are becoming popular. Dual and four-processor Pentium Pro motherboards, for example, are being used in workstations for CAD, image processing, visualization, and scientific computing. These systems offer high-end performance for less than the price of a conventional workstation. Furthermore many of the MPP systems today, have as their "nodes" 4-node multiprocessor systems.

There is, however, a problem with shared-memory threads and ANSI C++: the function pointers required for the thread interfaces are incompatible with templated and overloaded C++ functions. This is because it is hard to extract the address information of these types of functions.

Nearly all existing interface utilize function pointers in their thread-creation routines; for example, Windows NT uses the following declaration:

```
BOOL CreateThread(LPSECURITY_ATTRIBUTES secuirty_struct,
                  DWORD stack_size,
                  LPTHREAD_START_ROUTINE f,
                  LPVOID arg,
                  DWORD flag,
                  LPWORD threadID);
```

while POSIX threads utilize a similar declaration:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*entry)(void), void *arg);
```

A typical POSIX call would look like

```
    void f(const arg_struct *A);
    arg_struct Args;

    RTN = pthread_create(&TID, NULL, f, &Args);
```

The problem with overloaded functions in C++ is that there is no convenient way to specify a particular instantiation. For example, the following will not compile.

```
    int foo(int);
    int foo(double);
    int foo(complex);
    int foo(char);

    double Arg = 3.0;
    pthread_create(&TID, NULL, foo, &Arg);   //  which version of foo?
```

There is no mechanism to denote that we wish to spawn the `foo(double)` version , even though a specific `double` type is included in the argument list. The `pthread_create()` prototype essentially throws all type information away. Similarly, there are problems with templated functions because of their "instantiation on demand" characteristics:

```
    template <class T>
    int foo(const T& t){ ... };

    double Arg= 3.0;                         //  not possible to
    pthread_create(&TID, NULL, foo, &Arg);   //  to instantiate  foo(double);
```

4.4. Parallelism via BLAS kernels. Due to the problems with integrating threads with templated and overloaded functions, an alternative is to use this approach on conventional functions. A good candidate for such functions in TNT are the linear algebra kernels for sparse and dense matrices discussed in section 3.2.

For example, with matrix/vector multiply of a compressed-row sparse matrix with a dense vector, one can process group of rows in parallel. The basic operation is a vector dot product with a row of `A` with `x`. This is shown in Fig. 7. Each thread works on a panel of contiguous rows in `A`, with a similar partitioning of the result vector `y`. The `x` vector is shared (read-only) by all threads and may be effectively cached on a multi-processor system.

For compressed-column matrices, a different parallel algorithm is used. In this case, each thread works on a panel of consecutive columns of `A`, as shown in Fig. 8. The basic algorithm is a scalar vector update (`saxpy`) operation. The variable `x` is still shared by all threads and can also be efffectively cached. The values of `y`, however, need to be reduced over multiple threads, so some synchronization is needed to update these sums.

5. Conclusions. In the previous sections we have seen how resuable data structures and algorithms can be used in numerical linear algebra libraries. Much of the work presented here has borrowed ideas and has tried to be consistent with the paradigm of the Standard Template Library.

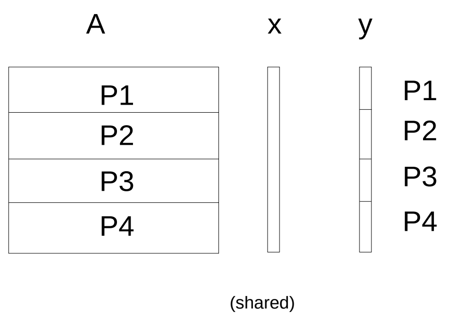

FIG. 7. *Parallel SMP algorithm for matrix/vector multiplication with compressed row matrices. The x vector is shared (read-only) by all threads and can effectively cached in a multiprocessor system.*

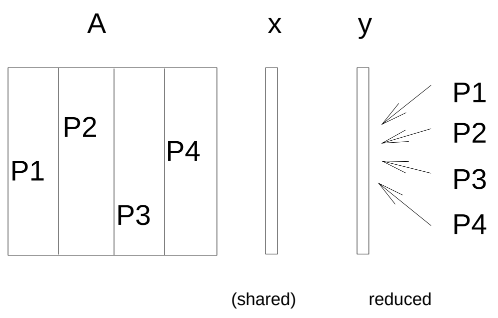

FIG. 8. *Parallel SMP algorithm for matrix/vector multiplication of compressed column matrices. The x vector is shared (read-only) by all threads and can effectively cached in a multiprocessor system. A reduction over y is needed.*

We have shown how one can write high-level numerical algorithms yet still generate executables with competitive performance to conventional C or Fortran libraries. The techniques in TNT utilize compile-time polymorphism in the form of C++ templates and overloaded functions. Thus, no excess run-time overhead is incurred due to virtual function lookup.

Although such an approach provides a practical platform for developing higher-level resuable components, the scheme is certainly not perfect. It does have some drawbacks. For instance, STL leads to code bloat since some linkers are not smart enough to fuse redundant function definitions when combining several object files. Programming with iterators can sometimes lead to difficult-to-track bugs. For example the code fragment to traverse an STL container "`for (p=a.begin(); p!=b.begin(); p++)...`" illustrates a small typo which can overwrite memory (`b.begin()` should be `a.begin()`). Furthermore STL does not deal with const correctness, and the STL template mechanism to specialize on templated function args is very fragile. (A good example of this are iterator tags).

For SMP parallel computing there are further challenges. Although portable thread-based are attractive, there are difficulties in using function-pointer OS interfaces with templated and overloaded functions in ANSI C++. The language extensions in HPC++ allow the simple pragmas to identify parallelizable loops. Such an approach can be used to annotate the algorithms in STL to produce a "generic" SMP library. HPC++ also defines an interface for a specific SMP version of STL; however variations of the basic STL algorithms, such as `par_for_each()` require recoding of application sources to utilize parallel versions. Thus it is as simple as recompiling and linking with a different flag option or library.

Another challenge for a parallel version of STL is that most of the non-random iterator based algorithms cannot be effectively parallelized due to their data dependencies.

To take advantage of small SMP platforms (from 1 to 8 processors), a parallel BLAS approach may prove most practical. By employing standard thread interfaces such as Windows NT and POSIX threads, we can implement such libraries and integrate them with application codes.

As mentioned in the introduction, this project is under constant development; this paper provides a look at the work in progress. Users are encouraged to visit the TNT web site, `http://math.nist.gov/tnt`, for latest development news and information.

REFERENCES

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, 1994.

[2] R. Boisvert, R. Pozo, K. Remington, "Matrix Market User's Guide", `http://math.nist.gov/MatrixMarket`.

[3] S. Carney, M. Heroux, G. Li, R. Pozo, K. Remington, K. Wu, "A Revised Proposal for a Sparse BLAS Tookit", `http://www.cray.com/PUBLIC/APPS/SERVICES/ALGORITHMS/spblastk.ps`

[4] J. Dongarra, J. Du Croz, I. S. Duff, S. Hammarling, "A set of level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Soft.*, Vol. 16, 1990, pp. 1-17.

[5] J. Dongarra, A. Lumsdaine, R. Pozo, K. Remington, *IML++: Iterative Methods Library Reference Guide*, NISTIR 5860, National Institute of Standards and Technology, `http://math.nist.gov/iml++`, June 1996.

[6] J. Dongarra, R. Pozo, D. Walker, "LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra," Proceedings of Supercomputing '93, IEEE Press, 1993, pp. 162-171.

[7] I. Duff, R. Grimes, J. Lewis, "Sparse Matrix Test Problems," *ACM Trans. Math. Soft.*, Vol. 15, 1989, pp. 1-14.

[8] I. Duff, M. Marrone, G. Radicati, *A Proposal for User Level Sparse BLAS*, CERFACS Technical Report TR/PA/92/85, 1992.

[9] HPC++ Working Group, *HPC++ Working Group White Paper*, `http://www.extreme.indiana.edu/hpc%2b%2b/docs/hpc++wp/hpc++wp.html`.

[10] E. Johnson, P. Beckman, D. Gannon, *HPC++: An experiment with the Parallel Standard Template Library*, `http://www.extreme.indiana.edu/hpc%2b%2b/docs/pstl/pstl.html`

[11] T. Q. Pham, P. K. Garg, *Multithreaded Programming with Windows NT*, Prentice-Hall, 1996.

[12] R. Pozo, *MV++ User's Manual*, NISTIR 5859, National Institute of Standards and Technology, `http://math.nist.gov/mv++`, June 1996.

[13] R. Pozo, K. Remington, A. Lumsdaine, *SparseLib++: Sparse Matrix Class Library Reference Guide*, NISTIR 5861, National Institute of Standards and Technology, `http://math.nist.gov/sparselib++`, June 1996.

[14] K. Remington, R. Pozo, "The NIST Sparse BLAS User's Guide", `http://math.nist.gov/spblas`.

[15] A. Stepanov, M. Lee, *The Standard Template Library*, Technical Report HPL-95-11, Hewlet-Packard Laboratories, January 1995.

[16] Sun Microsystems, Inc., *POSIX. 1c/D10 Summary*, 1995.